Virtual Memory

CS 351: Systems Programming
Michael Saelee <lee@iit.edu>
previously: SRAM ⇔ DRAM
next: DRAM ↔ HDD, SSD, etc.

i.e., memory as a “cache” for disk
main goals:

1. maximize memory throughput
2. maximize memory utilization
3. provide *address space consistency* & *memory protection* to processes
\[ \text{throughput} = \# \text{ bytes per second} \]

- depends on access latencies (DRAM, HDD) and “hit rate”
utilization = fraction of allocated memory that contains “user” data (aka payload)
- vs. metadata and other overhead required for memory management
address space consistency → provide a uniform “view” of memory to each process
address space consistency → provide a uniform “view” of memory to each process
memory protection → prevent processes from directly accessing each other’s address space
memory protection → prevent processes from directly accessing each other’s address space
i.e., every process should be provided with a managed, *virtualized* address space
“memory addresses”: what are they, really?
“physical” address: (byte) index into DRAM
int glob = 0xDEADBEEE;

main() {
    fork();
    glob += 1;
}

(gdb) set detach-on-fork off
(gdb) break main
Breakpoint 1 at 0x400508: file memtest.c, line 7.
(gdb) run
Breakpoint 1, main () at memtest.c:7
7    fork();
(gdb) next
[New process 7450]
8    glob += 1;
(gdb) print &glob
$1 = (int *) 0x6008d4
(gdb) next
9   }
(gdb) print /x glob
$2 = 0xdeadbeef
(gdb) inferior 2
[Switching to inferior 2 [process 7450]
#0  0x0000000310acac49d in __libc_fork ()
131       pid = ARCH_FORK ();
(gdb) finish
Run till exit from #0 in __libc_fork ()
8    glob += 1;
(gdb) print /x glob
$4 = 0xdeadbeee
(gdb) print &glob
$5 = (int *) 0x6008d4

parent

child
instructions executed by the CPU do not refer directly to *physical* addresses!
processes reference virtual addresses, the CPU relays virtual address requests to the memory management unit (MMU), which are translated to physical addresses
CPU

virtual address

MMU

address translation unit

physical address

disk address

Main Memory

“swap” space

(note: cache not shown)
essential problem: translate request for a virtual address $\rightarrow$ physical address

... this must be **FAST**, as *every* memory access from the CPU must be translated
both hardware/software are involved:

- MMU (hw) handles simple and fast operations (e.g., table lookups)

- Kernel (sw) handles complex tasks (e.g., eviction policy)
§ Virtual Memory Implementations
keep in mind goals:

1. maximize memory *throughput*
2. maximize memory *utilization*
3. provide *address space consistency* & *memory protection* to processes
1. simple relocation
1. simple relocation

- per-process relocation address is loaded by kernel on every context switch
1. simple relocation

- problem: processes may easily overextend their bounds and trample on each other
1. simple relocation

- incorporate a *limit* register to provide memory protection
1. simple relocation

- assertion failure triggers a fault, which summons kernel (which signals process)
pros:
- simple & fast!
- provides protection
but: available memory for mapping depends on value of base address
i.e., address spaces are *not consistent*!
also: all of a process *below the address limit* must be loaded in memory

i.e., memory may be *vastly under-utilized*
2. segmentation

- partition virtual address space into *multiple logical segments*

- individually map them onto physical memory with relocation registers
virtual address has form `seg#:offset`
assert (offset ≤ L₂)
- implemented as MMU registers
- part of kernel-maintained, per-process metadata (aka “process control block”)
- re-populated on each context switch
pros:

- still very \textit{fast}

- translation = register access \& addition

- memory \textit{protection} via limits

- segmented addresses improve \textit{consistency}
simple relocation:

segmentation:
- variable segment sizes → memory fragmentation
- fragmentation potentially lowers utilization
- can fix through compaction, but expensive!
3. paging

- partition virtual and physical address spaces into *uniformly sized* pages

- virtual pages map onto physical pages
- minimum mapping granularity = page
- not all of a given segment need be mapped
modified mapping problem:

- a virtual address is broken down into virtual page number & page offset

- determine which physical page (if any) a given virtual page is loaded into

- if physical page is found, use page offset to access data
Given page size = $2^p$ bytes

VA: \[\text{virtual page number} \quad \text{virtual page offset}\]

PA: \[\text{physical page number} \quad \text{physical page offset}\]
VA:

<table>
<thead>
<tr>
<th>virtual page number</th>
<th>virtual page offset</th>
</tr>
</thead>
</table>

PA:

<table>
<thead>
<tr>
<th>physical page number</th>
<th>physical page offset</th>
</tr>
</thead>
</table>

address translation
**translation structure: page table**

- **VA:** virtual page number \( \rightarrow \) virtual page offset
- **PA:** physical page number \( \rightarrow \) physical page offset

- **index**
  - If invalid, page is not mapped

- **valid**
  - PPN

- **2^n entries**
page table entries (PTEs) typically contain additional metadata, e.g.:

- dirty (modified) bit
- access bits (shared or kernel-owned pages may be read-only or inaccessible)
e.g., 32-bit virtual address, 4KB ($2^{12}$) page size, 4-byte PTE size;
- size of page table?
e.g., 32-bit virtual address, 4KB \(2^{12}\) pages, 4-byte PTEs;

- \# pages = \(2^{32} \div 2^{12} = 2^{20} = 1\text{M}\)

- page table size = \(1\text{M} \times 4\text{ bytes} = 4\text{MB}\)
4MB is much too large to fit in the MMU — insufficient registers and SRAM!

Page table resides in **main memory**
The translation process (aka *page table walk*) is performed by hardware (MMU).

The kernel must initially populate, then continue to manage a process’s page table.

The kernel also populates a *page table base register* on context switches.
translation: \emph{hit}

1. VA: $N$
2. page table walk
3. PA: $N'$
4. data
translation: *miss*

1. **VA: N**
2. **page table walk**
3. **page fault**
4. **transfer control to kernel**
5. **data transfer**
6. **PTE update**
7. **VA: N (retry)**
8. **walk**
9. **PA: N'**
10. **data**

**CPU**

**Main Memory**

**Page Table**

**Address Translator (part of MMU)**

**kernel**

**Disk (swap space)**
kernel decides where to place page, and what to evict (if memory is full)

- e.g., using LRU replacement policy
this system enables **on-demand paging**
i.e., an active process need only be partly in memory (load rest from disk dynamically)
but if working set (of active processes) exceeds available memory, we may have **swap thrashing**
integration with caches?
Q: do caches use physical or virtual addresses for lookups?
Virtual address based Cache

Process A

Virtual Address Space

Virtual Address Space

Process B

CPU

Cache

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>X</td>
</tr>
<tr>
<td>M</td>
<td>Y</td>
</tr>
<tr>
<td>N</td>
<td>Z</td>
</tr>
</tbody>
</table>

ambiguous!
Physical address based Cache

Process A

Virtual Address Space

Process B

Virtual Address Space

CPU

Cache

Address | Data
---|---
S | X
Q | Y
R | Z

Physical Memory

X
Z
Y
Q: do caches use physical or virtual addresses for lookups?

A: caches typically use physical addresses
%o*@$&#!!!
saved by hardware:

the *Translation Lookaside Buffer* (TLB) — a cache used solely for VPN→PPN lookups
TLB + Page table
(exercise for reader: revise earlier translation diagrams!)
virtual address

virtual page number (VPN)  page offset

valid  tag  physical page number (PPN)

physical address

TLB

valid  tag  data

byte offset

Cache

TLB Hit

Cache Hit
TLB mappings are *process specific* — requires flush & reload on context switch

- some architectures store PID (aka “virtual space” ID) in TLB
Familiar caching problem:
- TLB caches a few thousand mappings
- vs. millions of virtual pages per process!
we can improve TLB hit rate by reducing the number of pages ... by increasing the size of each page
compute # pages for 32-bit memory for:

- 1KB, 512KB, 4MB pages
  - $2^{32} \div 2^{10} = 2^{22} = 4M$ pages
  - $2^{32} \div 2^{19} = 2^{13} = 8K$ pages
  - $2^{32} \div 2^{22} = 2^{10} = 1K$ pages

(not bad!)
Process A

Virtual Memory

Physical Memory

Virtual Memory

 lots of wasted space!

Process B

Virtual Memory
increasing page size results in increased internal fragmentation and lower utilization
i.e., TLB effectiveness needs to be balanced against memory utilization
so what about 64-bit systems?

\[ 2^{64} = 16 \text{ Exabyte address space} \approx 4 \text{ billion } \times 4 \text{GB} \]
most modern implementations support a max of $2^{48}$ (256TB) addressable space
page table size (assuming 4K page size)?

- # pages \( = \frac{2^{48}}{2^{12}} = 2^{36} \)
- PTE size \( = 8 \text{ bytes (64 bits)} \)
- PT size \( = 2^{36} \times 8 = 2^{39} \text{ bytes} \)

\[ = 512\text{GB} \]
512GB
(just for the virtual memory *mapping* structure)
(and we need *one per process*)
(these things aren’t going to fit in memory)
instead, use *multi-level* page tables:

- split an address translation into two (or more) separate table lookups

- unused parts of the table don’t need to be in memory!
“toy” memory system
- 8 bit addresses
- 32-byte pages

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

- VPN
- page offset

Page Table
- all 8 PTEs must be in memory at all times
“toy” memory system
- 8 bit addresses
- 32-byte pages

page "directory"
all unmapped; don’t need in memory!
“toy” memory system
- 8 bit addresses
- 32-byte pages
Intel Architecture Memory Management

http://www.intel.com/products/processor/manuals/

(Software Developer’s Manual Volume 3A)
segment, the segment type, and the location of the first byte of the segment in the linear address space (called the base address of the segment). The offset part of the logical address is added to the base address for the segment to locate a byte within the segment. The base address plus the offset thus forms a linear address in the processor's linear address space.

If paging is not used, the linear address space of the processor is mapped directly into the physical address space of the processor. The physical address space is defined as the range of addresses that the processor can generate on its address bus. Because multitasking computing systems commonly define a linear address space much larger than it is economically feasible to contain all at once in physical memory, some method of "virtualizing" the linear address space is needed. This virtualization of the linear address space is handled through the processor's paging mechanism. Paging supports a "virtual memory" environment where a large linear address space is simulated with a small amount of physical memory (RAM and ROM) and some disk.
PROTECTED-MODE MEMORY MANAGEMENT

If paging is not used, the processor maps the linear address directly to a physical address (that is, the linear address goes out on the processor’s address bus). If the linear address space is paged, a second level of address translation is used to translate the linear address into a physical address.

See also: Chapter 4, “Paging.”

3.4.1 Logical Address Translation in IA-32e Mode

In IA-32e mode, an Intel 64 processor uses the steps described above to translate a logical address to a linear address. In 64-bit mode, the offset and base address of the segment are 64-bits instead of 32 bits. The linear address format is also 64 bits wide and is subject to the canonical form requirement.

Each code segment descriptor provides an L bit. This bit allows a code segment to execute 64-bit code or legacy 32-bit code by code segment.

3.4.2 Segment Selectors

A segment selector is a 16-bit identifier for a segment (see Figure 3-6). It does not point directly to the segment, but instead points to the segment descriptor that defines the segment. A segment selector contains the following items:

Index (Bits 3 through 15) — Selects one of 8192 descriptors in the GDT or LDT. The processor multiplies the index value by 8 (the number of bytes in a segment descriptor) and adds the result to the base address of the GDT or LDT (from the GDTR or LDTR register, respectively).

TI (table indicator) flag (Bit 2) — Specifies the descriptor table to use: clearing this flag selects the GDT; setting this flag selects the current LDT.

Figure 3-5. Logical Address to Linear Address Translation

Figure 3-6. Segment Selector

Segmented → Linear Address
Segment registers

<table>
<thead>
<tr>
<th>Visible Part</th>
<th>Hidden Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment Selector</td>
<td>Base Address, Limit, Access Information</td>
</tr>
<tr>
<td>CS</td>
<td></td>
</tr>
<tr>
<td>SS</td>
<td></td>
</tr>
<tr>
<td>DS</td>
<td></td>
</tr>
<tr>
<td>ES</td>
<td></td>
</tr>
<tr>
<td>FS</td>
<td></td>
</tr>
<tr>
<td>GS</td>
<td></td>
</tr>
</tbody>
</table>
The flags and fields in a segment descriptor are as follows:

**Segment limit field**
Specifies the size of the segment. The processor puts together the two segment limit fields to form a 20-bit value. The processor interprets the segment limit in one of two ways, depending on the setting of the G (granularity) flag:

- If the granularity flag is clear, the segment size can range from 1 byte to 1 MByte, in byte increments.
- If the granularity flag is set, the segment size can range from 4 KBytes to 4 GBytes, in 4-KByte increments.

The processor uses the segment limit in two different ways, depending on whether the segment is an expand-up or an expand-down segment. See Section 3.4.5.1, "Code- and Data-Segment Descriptor Types", for more information about segment types. For expand-up segments, the offset in a logical address can range from 0 to the segment limit. Offsets greater than the segment limit generate general-protection exceptions (#GP, for all segments other than SS) or stack-fault exceptions (#SS for the SS segment). For expand-down segments, the segment limit has the reverse function; the offset can range from the segment limit plus 1 to FFFFFFFFH or FFFFH, depending on the setting of the B flag. Offsets less than or equal to the segment limit generate general-protection exceptions or stack-fault exceptions. Decreasing the value in the segment limit field for an expand-down segment allocates new memory at the bottom of the segment's address space, rather than at the top. IA-32 architecture stacks always grow downwards, making this mechanism convenient for expandable stacks.

**Base address fields**
Defines the location of byte 0 of the segment within the 4-GByte linear address space. The processor puts together the three base address fields to form a single 32-bit value. Segment base addresses should be aligned to 16-byte boundaries. Although 16-byte alignment is not required, this alignment allows programs to maximize performance by aligning code and data on 16-byte boundaries.

**Type field**
Indicates the segment or gate type and specifies the kinds of access that can be made to the segment and the direction of growth. The interpretation of this field depends on whether the descriptor type flag specifies an application (code or data) descriptor or a system descriptor. The encoding of the type field is different for code, data, and system descriptors (see Figure 5-1). See Section 3.4.5.1, "Code- and Data-Segment Descriptor Types", for a description of how this field is used to specify code and data-segment types.

### Figure 3-8. Segment Descriptor

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| G  | D  | /B | L  | AVL| Seg. Limit 19:16 | P  | DPL| S  | Type | Base 31:24 | 24 23 22 21 20 | 19 18 17 16 15 | 14 13 12 11 | 8 7 | 6 5 | 4 | 3 | 2 | 1 | 0 |
| L  | AVL| BASE| D/B| DPL| TYPE | AVL| BASE| D/B| DPL| TYPE | BASE | D/B | DPL| TYPE | D/B | DPL| TYPE | BASE | D/B | DPL| TYPE | BASE | D/B | DPL| TYPE | BASE | D/B | DPL| TYPE |

- **L** — 64-bit code segment (IA-32e mode only)
- **AVL** — Available for use by system software
- **BASE** — Segment base address
- **D/B** — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
- **DPL** — Descriptor privilege level
- **G** — Granularity
- **LIMIT** — Segment Limit
- **P** — Segment present
- **S** — Descriptor type (0 = system; 1 = code or data)
- **TYPE** — Segment type

**Segment descriptor**
Access checks can be used to protect not only against referencing an address outside the limit of a segment, but also against performing disallowed operations in certain segments. For example, since code segments are designated as read-only segments, hardware can be used to prevent writes into code segments. The access rights information created for segments can also be used to set up protection rings or levels. Protection levels can be used to protect operating-system procedures from unauthorized access by application programs.

3.2.4 Segmentation in IA-32e Mode

In IA-32e mode of Intel 64 architecture, the effects of segmentation depend on whether the processor is running in compatibility mode or 64-bit mode. In compatibility mode, segmentation functions just as it does using legacy 16-bit or 32-bit protected mode semantics.

Segmented address space
PROTECTED-MODE MEMORY MANAGEMENT

3.2.2 Protected Flat Model

The protected flat model is similar to the basic flat model, except the segment limits are set to include only the range of addresses for which physical memory actually exists (see Figure 3-3). A general-protection exception (#GP) is then generated on any attempt to access nonexistent memory. This model provides a minimum level of hardware protection against some kinds of program bugs.

“Flat” address space

FFFF_FFF0H. RAM (DRAM) is placed at the bottom of the address space because the initial base address for the DS data segment after reset initialization is 0.
Table 4-1 illustrates the key differences between the three paging modes.

Because they are used only if IA32_EFER.LME = 0, 32-bit paging and PAE paging is used only in legacy protected mode. Because legacy protected mode cannot produce linear addresses larger than 32 bits, 32-bit paging and PAE paging translate 32-bit linear addresses.

Because it is used only if IA32_EFER.LME = 1, IA-32e paging is used only in IA-32e mode. (In fact, it is the use of IA-32e paging that defines IA-32e mode.) IA-32e mode has two sub-modes:

- **Compatibility mode.** This mode uses only 32-bit linear addresses. IA-32e paging treats bits 47:32 of such an address as all 0.
- **64-bit mode.** While this mode produces 64-bit linear addresses, the processor ensures that bits 63:47 of such an address are identical.

**Table 4-1. Properties of Different Paging Modes**

<table>
<thead>
<tr>
<th>Paging Mode</th>
<th>CR0.PG</th>
<th>CR4.PAE</th>
<th>LME in IA32_EFER</th>
<th>Linear-Address Width</th>
<th>Physical-Address Width</th>
<th>Page Size(s)</th>
<th>Supports Execute-Disable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0</td>
<td>N/A</td>
<td>N/A</td>
<td>32</td>
<td>32</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>32-bit</td>
<td>1</td>
<td>0</td>
<td>0²</td>
<td>32</td>
<td>Up to 40³</td>
<td>4-KByte 4-MByte</td>
<td>No</td>
</tr>
<tr>
<td>PAE</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>32</td>
<td>Up to 52</td>
<td>4-KByte 2-MByte</td>
<td>Yes⁵</td>
</tr>
<tr>
<td>IA-32e</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>48</td>
<td>Up to 52</td>
<td>4-KByte 2-MByte 1-GByte</td>
<td>Yes⁵</td>
</tr>
</tbody>
</table>

**NOTES:**

1. The physical-address width is always bounded by MAXPHYADDR; see Section 4.1.4.
2. The processor ensures that IA32_EFER.LME must be 0 if CR0.PG = 1 and CR4.PAE = 0.
3. 32-bit paging supports physical-address widths of more than 32 bits only for 4-MByte pages and only if the PSE-36 mechanism is supported; see Section 4.1.4 and Section 4.3.
4. 4-MByte pages are used with 32-bit paging only if CR4.PSE = 1; see Section 4.3.
5. Execute-disable access rights are applied only if IA32_EFER.NXE = 1; see Section 4.6.
6. Not all processors that support IA-32e paging support 1-GByte pages; see Section 4.1.4.

**Paging modes**

1. Such an address is called *canonical*. Use of a non-canonical linear address in 64-bit mode produces a general-protection exception (#GP(0)); the processor does not attempt to translate non-canonical linear addresses using IA-32e paging.
IA-32 paging (4KB pages)
IA-32 paging (4MB pages)
PTE formats (32-bit paging)

<table>
<thead>
<tr>
<th>Address of page directory(^1)</th>
<th>Ignored</th>
<th>(\text{PC})</th>
<th>(\text{PD})</th>
<th>(\text{PT})</th>
<th>Ignored</th>
<th>CR3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits 31:22 of address of 2MB page frame</td>
<td>Reserved (must be 0)</td>
<td>Bits 39:32 of address(^2)</td>
<td>(\text{PAT})</td>
<td>Ignored</td>
<td>(\text{G})</td>
<td>(\text{1})</td>
</tr>
<tr>
<td>Address of page table</td>
<td>Ignored</td>
<td>(\text{O})</td>
<td>Ignored</td>
<td>(\text{A})</td>
<td>(\text{PC})</td>
<td>(\text{PD})</td>
</tr>
<tr>
<td>Ignored</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address of 4KB page frame</td>
<td>Ignored</td>
<td>(\text{G})</td>
<td>(\text{PAT})</td>
<td>(\text{DA})</td>
<td>(\text{PC})</td>
<td>(\text{PD})</td>
</tr>
<tr>
<td>Ignored</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
PAE paging (4KB pages)
IA-32e paging (4KB pages)
The following items describe the IA-32e paging process in more detail as well as how the page size is determined.

• A 4-KByte naturally aligned PML4 table is located at the physical address specified in bits 51:12 of CR3 (see Table 4-12). A PML4 table comprises 512 64-bit entries (PML4Es). A PML4E is selected using the physical address defined as follows:
  - Bits 51:12 are from CR3.
  - Bits 11:3 are bits 47:39 of the linear address.
  - Bits 2:0 are all 0.

Because a PML4E is identified using bits 47:39 of the linear address, it controls access to a 512-GByte region of the linear-address space.

• A 4-KByte naturally aligned page-directory-pointer table is located at the physical address specified in bits 51:12 of the PML4E (see Table 4-14). A page-directory-pointer table comprises 512 64-bit entries (PDPTEs). A PDPTE is selected using the physical address defined as follows:
  - Bits 51:12 are from the PML4E.

Figure 4-10. Linear-Address Translation to a 1-GByte Page using IA-32e Paging

IA-32e paging (1GB pages)